下载APP

17 | 内存视角（三）：OOM都是谁的锅？怎么破？

2021-04-21 吴磊

Spark性能调优实战

进入课程 >

讲述：吴磊

时长 17:39　大小 16.17M

你好，我是吴磊。

无论是批处理、流计算，还是数据分析、机器学习，只要是在 Spark 作业中，我们总能见到 OOM（Out Of Memory，内存溢出）的身影。一旦出现 OOM，作业就会中断，应用的业务功能也都无法执行。因此，及时处理 OOM 问题是我们日常开发中一项非常重要的工作。

但是，Spark 报出的 OOM 问题可以说是五花八门，常常让人找不到头绪。比如，我们经常遇到，数据集按照尺寸估算本该可以完全放进内存，但 Spark 依然会报 OOM 异常。这个时候，不少同学都会参考网上的做法，把 spark.executor.memory 不断地调大、调大、再调大，直到内心崩溃也无济于事，最后只能放弃。

那么，当我们拿到 OOM 这个“烫手的山芋”的时候该怎么办呢？我们最先应该弄清楚的是“**到底哪里出现了 OOM**”。只有准确定位出现问题的具体区域，我们的调优才能有的放矢。具体来说，这个“**哪里**”，我们至少要分 3 个方面去看。

发生 OOM 的 LOC（Line Of Code），也就是代码位置在哪?

OOM 发生在 Driver 端，还是在 Executor 端?

如果是发生在 Executor 端，OOM 到底发生在哪一片内存区域?

定位出错代码的位置非常重要但也非常简单，我们只要利用 Stack Trace 就能很快找到抛出问题的 LOC。因此，更**关键的是，我们要明确出问题的到底是 Driver 端还是 Executor 端，以及是哪片内存区域**。Driver 和 Executor 产生 OOM 的病灶不同，我们自然需要区别对待。

所以今天这一讲，我们就先来说说 Driver 端的 OOM 问题和应对方法。由于内存在 Executor 端被划分成了不同区域，因此，对于 Executor 端怪相百出的 OOM，我们还要结合案例来分类讨论。最后，我会带你整理出一套应对 OOM 的“武功秘籍”，让你在面对 OOM 的时候，能够见招拆招、有的放矢！

Driver 端的 OOM

我们先来说说 Driver 端的 OOM。Driver 的主要职责是任务调度，同时参与非常少量的任务计算，因此 Driver 的内存配置一般都偏低，也没有更加细分的内存区域。

因为 Driver 的内存就是囫囵的那么一块，所以 Driver 端的 OOM 问题自然不是调度系统的毛病，只可能来自它涉及的计算任务，主要有两类：

创建小规模的分布式数据集：使用 parallelize、createDataFrame 等 API 创建数据集

收集计算结果：通过 take、show、collect 等算子把结果收集到 Driver 端

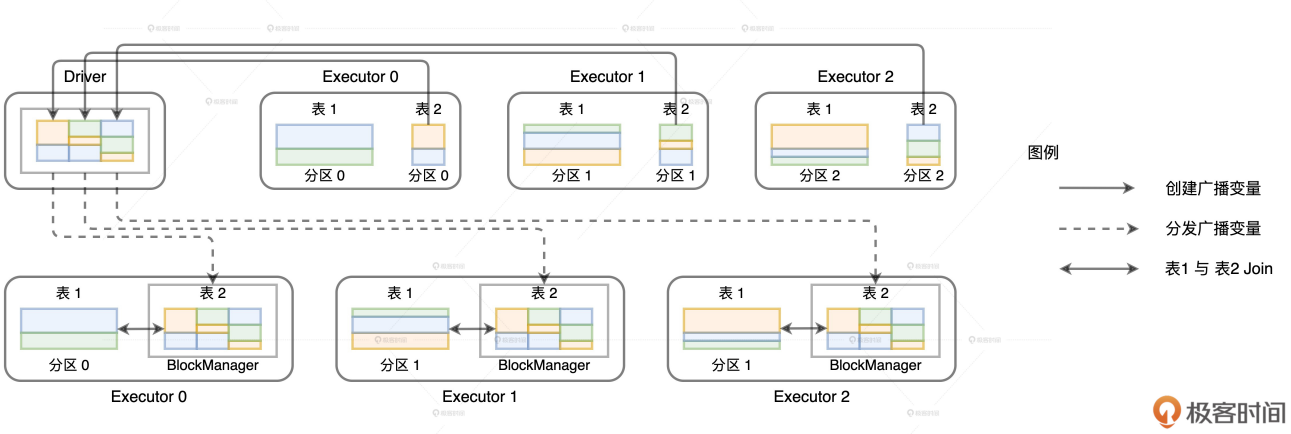
因此 Driver 端的 OOM 逃不出 2 类病灶：

创建的数据集超过内存上限

收集的结果集超过内存上限

第一类病灶不言自明，咱们不细说了。看到第二类病灶，想必你第一时间想到的就是万恶的 collect。确实，说到 OOM 就不得不提 collect。collect 算子会从 Executors 把全量数据拉回到 Driver 端，因此，如果结果集尺寸超过 Driver 内存上限，它自然会报 OOM。

由开发者直接调用 collect 算子而触发的 OOM 问题其实很好定位，比较难定位的是间接调用 collect 而导致的 OOM。那么，间接调用 collect 是指什么呢？还记得广播变量的工作原理吗？



广播变量的创建与分发

广播变量在创建的过程中，需要先把分布在所有 Executors 的数据分片拉取到 Driver 端，然后在 Driver 端构建广播变量，最后 Driver 端把封装好的广播变量再分发给各个 Executors。第一步的数据拉取其实就是用 collect 实现的。如果 Executors 中数据分片的总大小超过 Driver 端内存上限也会报 OOM。在日常的调优工作中，你看到的表象和症状可能是：

```
1 java.lang.OutOfMemoryError: Not enough memory to build and broadcast
```

但实际的病理却是 Driver 端内存受限，没有办法容纳拉取回的结果集。找到了病因，再去应对 Driver 端的 OOM 就很简单了。我们只要对结果集尺寸做适当的预估，然后再相应地增加 Driver 侧的内存配置就好了。调节 Driver 端侧内存大小我们要用到 spark.driver.memory 配置项，预估数据集尺寸可以用“先 Cache，再查看执行计划”的方式，示例代码如下。

```
1 val df: DataFrame = _
```

```
df.cache.count
val plan = df.queryExecution.logical
val estimated: BigInt = spark
.sessionState
.executePlan(plan)
.optimizedPlan
.stats
.sizeInBytes
```

Executor 端的 OOM

我们再来说说 Executor 端的 OOM。我们知道，执行内存分为 4 个区域：Reserved Memory、User Memory、Storage Memory 和 Execution Memory。这 4 个区域中都有哪些区域会报 OOM 异常呢？哪些区域压根就不存在 OOM 的可能呢？

在 Executors 中，与任务执行有关的内存区域才存在 OOM 的隐患。其中，Reserved Memory 大小固定为 300MB，因为它是硬编码到源码中的，所以不受用户控制。而对于 Storage Memory 来说，即便数据集不能完全缓存到 MemoryStore，Spark 也不会抛 OOM 异常，额外的数据要么落盘（MEMORY_AND_DISK）、要么直接放弃（MEMORY_ONLY）。

因此，当 Executors 出现 OOM 的问题，我们可以先把 Reserved Memory 和 Storage Memory 排除，然后锁定 Execution Memory 和 User Memory 去找毛病。

User Memory 的 OOM

在内存管理那一讲，我们说过 User Memory 用于存储用户自定义的数据结构，如数组、列表、字典等。因此，如果这些数据结构的总大小超出了 User Memory 内存区域的上限，你可能就会看到下表示例中的报错。

复制代码

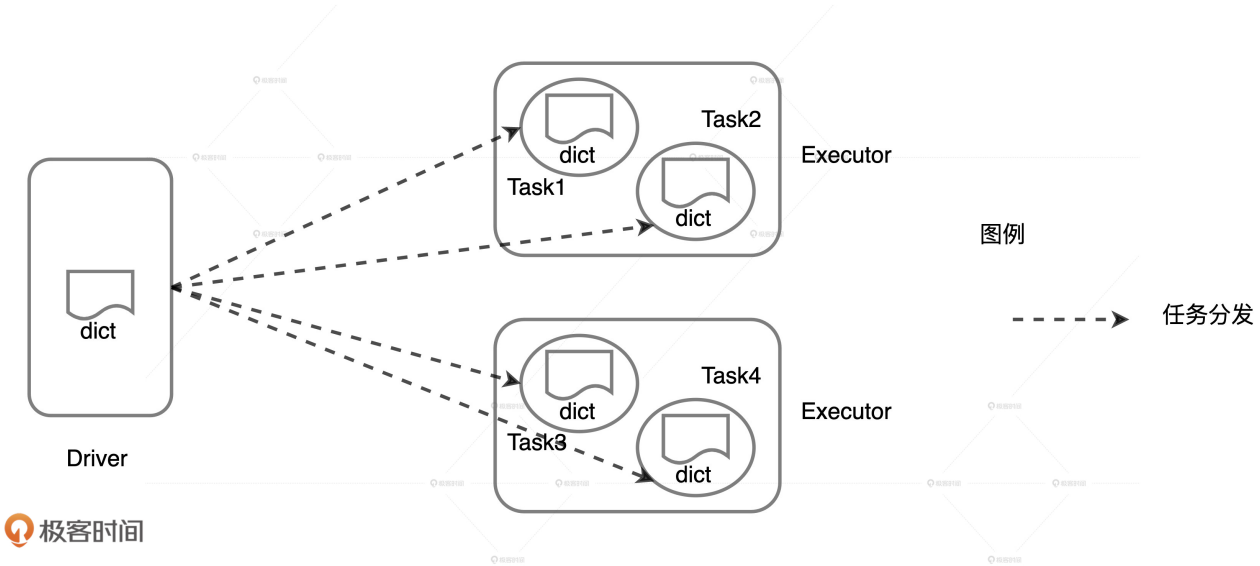
```
java.lang.OutOfMemoryError: Java heap space at java.util.Arrays.copyOf

java.lang.OutOfMemoryError: Java heap space at java.lang.reflect.Array.newInst
```

如果你的数据结构是用于分布式数据转换，在计算 User Memory 内存消耗时，你就需要考虑 Executor 的线程池大小。还记得下面的这个例子吗？

复制代码

```
val dict = List(“spark”, “tune”)
val words = spark.sparkContext.textFile(“~/words.csv”)
val keywords = words.filter(word => dict.contains(word))
keywords.map((_, 1)).reduceByKey(_ + _).collect
```

自定义的列表 dict 会随着 Task 分发到所有 Executors，因此多个 Task 中的 dict 会对 User Memory 产生重复消耗。如果把 dict 尺寸记为 #size，Executor 线程池大小记为 #threads，那么 dict 对 User Memory 的总消耗就是：#size * #threads。一旦总消耗超出 User Memory 内存上限，自然就会产生 OOM 问题。



用户数据在任务中的分发

那么，解决 User Memory 端 OOM 的思路和 Driver 端的并无二致，也是先对数据结构的消耗进行预估，然后相应地扩大 User Memory 的内存配置。不过，相比 Driver，User Memory 内存上限的影响因素更多，总大小由 spark.executor.memory * （1 - spark.memory.fraction）计算得到。

Execution Memory 的 OOM

要说 OOM 的高发区，非 Execution Memory 莫属。久行夜路必撞鬼，在分布式任务执行的过程中，Execution Memory 首当其冲，因此出错的概率相比其他内存区域更高。关于 Execution Memory 的 OOM，我发现不少同学都存在这么一个误区：只要数据量比执行内存小就不会发生 OOM，相反就会有一定的几率触发 OOM 问题。

实际上，**数据量并不是决定 OOM 与否的关键因素，数据分布与 Execution Memory 的运行时规划是否匹配才是**。这么说可能比较抽象，你还记得黄小乙的如意算盘吗？为了提高老乡们种地的热情和积极性，他制定了个转让协议，所有老乡申请的土地面积介于 1/N/2 和 1/N 之间。因此，如果有的老乡贪多求快，买的种子远远超过 1/N 上限能够容纳的数量，这位老乡多买的那部分种子都会被白白浪费掉。

同样的，我们可以把 Execution Memory 看作是

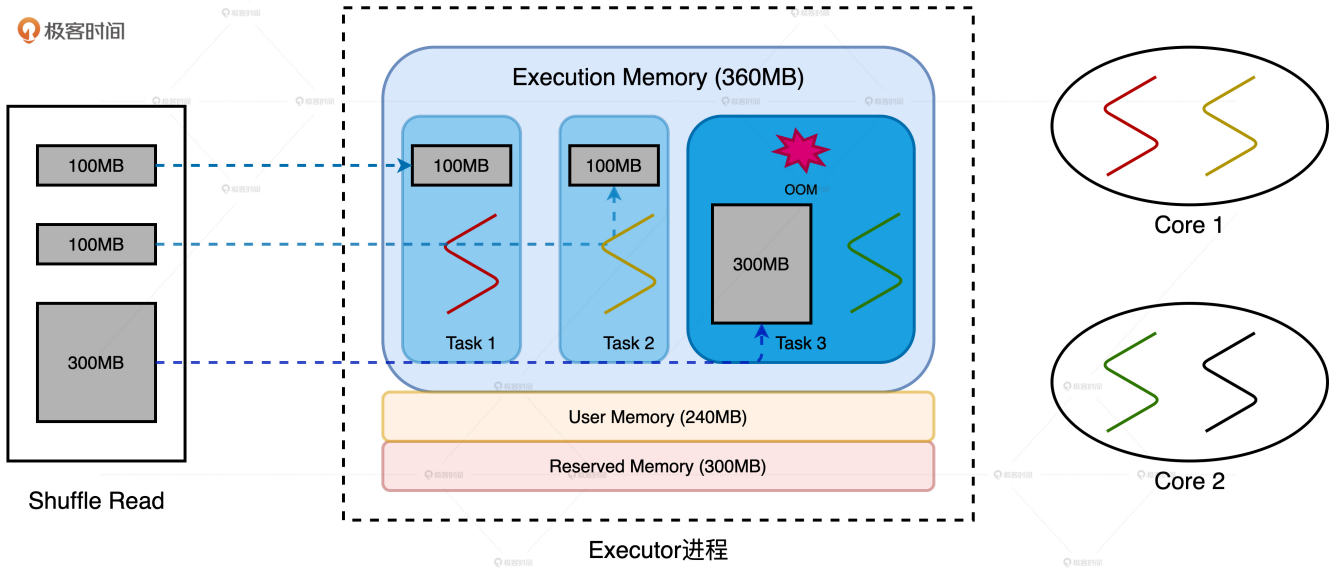
土地，把分布式数据集看作是种子，一旦**分布式任务的内存请求超出 1/N 这个上限，**Execution Memory 就会出现 OOM 问题。而且，相比其他场景下的 OOM 问题，Execution Memory 的 OOM 要复杂得多，它不仅仅与内存空间大小、数据分布有关，还与 Executor 线程池和运行时任务调度有关。

抓住了引起 OOM 问题最核心的原因，对于 Execution Memory OOM 的诸多表象，我们就能从容应对了。下面，我们就来看两个平时开发中常见的实例：数据倾斜和数据膨胀。为了方便说明，在这两个实例中，计算节点的硬件配置是一样的，都是 2 个 CPU core，每个 core 有两个线程，内存大小为 1GB，并且 spark.executor.cores 设置为 3，spark.executor.memory 设置为 900MB。

根据配置项那一讲我们说过的不同内存区域的计算公式，在默认配置下，我们不难算出 Execution Memory 和 Storage Memory 内存空间都是 180MB。而且，因为我们的例子里没有 RDD 缓存，所以 Execution Memory 内存空间上限是 360MB。

实例 1：数据倾斜

我们先来看第一个数据倾斜的例子。节点在 Reduce 阶段拉取数据分片，3 个 Reduce Task 对应的数据分片大小分别是 100MB 和 300MB。显然，第三个数据分片存在轻微的数据倾斜。由于 Executor 线程池大小为 3，因此每个 Reduce Task 最多可获得 360MB * 1 / 3 = 120MB 的内存空间。Task1、Task2 获取到的内存空间足以容纳分片 1、分片 2，因此可以顺利完成任务。



数据倾斜导致OOM

Task3 的数据分片大小远超内存上限，即便 Spark 在 Reduce 阶段支持 Spill 和外排，120MB 的内存空间也无法满足 300MB 数据最基本的计算需要，如 PairBuffer 和 AppendOnlyMap 等数据结构的内存消耗，以及数据排序的临时内存消耗等等。

这个例子的表象是数据倾斜导致 OOM，但实质上是 Task3 的内存请求超出 1/N 上限。因此，针对以这个案例为代表的数据倾斜问题，我们至少有 2 种调优思路：

消除数据倾斜，让所有的数据分片尺寸都不大于 100MB

调整 Executor 线程池、内存、并行度等相关配置，提高 1/N 上限到 300MB

每一种思路都可以衍生出许多不同的方法，就拿第 2 种思路来说，要满足 1/N 的上限，最简单地，我们可以把 spark.executor.cores 设置成 1，也就是 Executor 线程池只有一个线程“并行”工作。这个时候，每个任务的内存上限都变成了 360MB，容纳 300MB 的数据分片绰绰有余。

当然，线程池大小设置为 1 是不可取的，刚刚只是为了说明调优的灵活性。延续第二个思路，你需要去平衡多个方面的配置项，在充分利用 CPU 的前提下解决 OOM 的问题。比如：

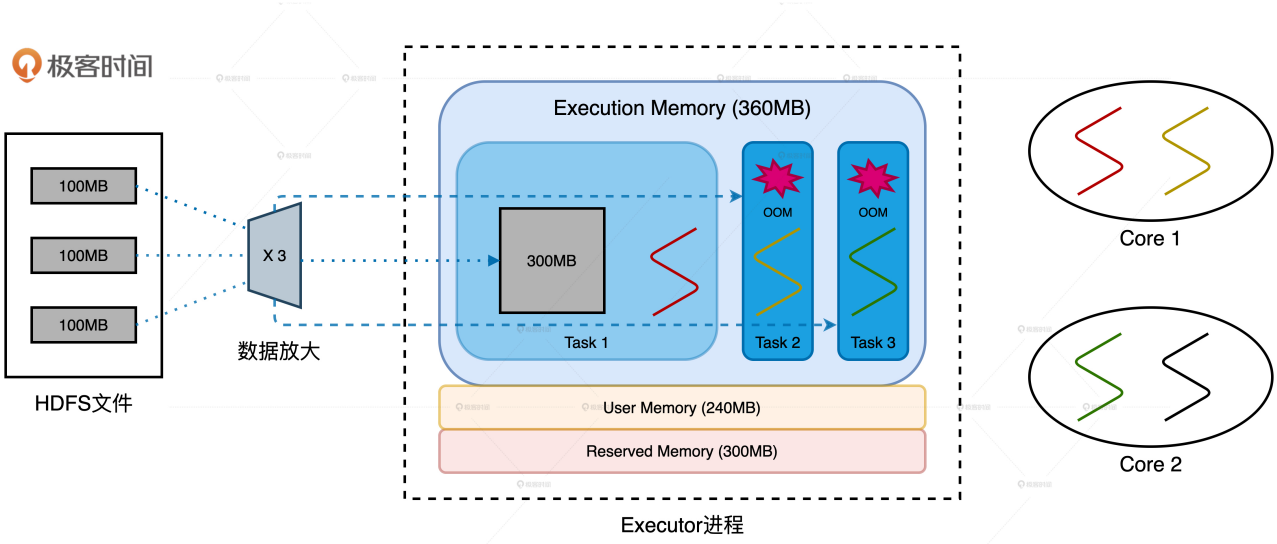
维持并发度、并行度不变，增大执行内存设置，提高 1/N 上限到 300MB

维持并发度、执行内存不变，使用相关配置项来提升并行度将数据打散，让所有的数据分片尺寸都缩小到 100MB 以内

关于线程池、内存和并行度之间的平衡与设置，我在 CPU 视角那一讲做过详细的介绍，你可以去回顾一下。至于怎么消除数据倾斜，你可以好好想想，再把你的思路分享出来。

实例 2：数据膨胀

我们再来看第二个数据膨胀的例子。节点在 Map 阶段拉取 HDFS 数据分片，3 个 Map Task 对应的数据分片大小都是 100MB。按照之前的计算，每个 Map Task 最多可获得 120MB 的执行内存，不应该出现 OOM 问题才对。



数据膨胀导致OOM

尴尬的地方在于，磁盘中的数据进了 JVM 之后会膨胀。在我们的例子中，数据分片加载到 JVM Heap 之后翻了 3 倍，原本 100MB 的数据变成了 300MB，因此，OOM 就成了一件必然会发生的事情。

在这个案例中，表象是数据膨胀导致 OOM，但本质上还是 Task2 和 Task3 的内存请求超出 1/N 上限。因此，针对以这个案例为代表的数据膨胀问题，我们还是有至少 2 种调优思路：

把数据打散，提高数据分片数量、降低数据粒度，让膨胀之后的数据量降到 100MB 左右

加大内存配置，结合 Executor 线程池调整，提高 1/N 上限到 300MB

小结

想要高效解决五花八门的 OOM 问题，最重要的就是准确定位问题出现的区域，这样我们的调优才能有的放矢，我建议你按照两步进行。

首先，定位 OOM 发生的代码位置，你通过 Stack Trace 就能很快得到答案。

其次，定位 OOM 是发生在 Driver 端还是在 Executor 端。如果是发生在 Executor 端，再定位具体发生的区域。

发生在 Driver 端的 OOM 可以归结为两类：

创建的数据集超过内存上限

收集的结果集超过内存上限

应对 Driver 端 OOM 的常规方法，是先适当预估结果集尺寸，然后再相应增加 Driver 侧的内存配置。

发生在 Executors 侧的 OOM 只和 User Memory 和 Execution Memory 区域有关，因为它们都和任务执行有关。其中，User Memory 区域 OOM 的产生的原因和解决办法与 Driver 别无二致，你可以直接参考。

而 Execution Memory 区域 OOM 的产生的原因是数据分布与 Execution Memory 的运行时规划不匹配，也就是分布式任务的内存请求超出了 1/N 上限。解决 Execution Memory 区域 OOM 问题的思路总的来说可以分为 3 类：

消除数据倾斜，让所有的数据分片尺寸都小于 1/N 上限

把数据打散，提高数据分片数量、降低数据粒度，让膨胀之后的数据量降到 1/N 以下

加大内存配置，结合 Executor 线程池调整，提高 1/N 上限

每日一练

1. 数据膨胀导致 OOM 的例子中，为什么 Task1 能获取到 300MB 的内存空间？（提示：可以回顾 CPU 视角那一讲去寻找答案。）

2. 在日常开发中，你还遇到过哪些 OOM 表象？你能把它们归纳到我们今天讲的分类中吗？

期待在留言区看到你的思考和分享，我们下一讲见！

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 内存视角（二）：如何有效避免Cache滥用？

下一篇 18 | 磁盘视角：如果内存无限大，磁盘还有用武之地吗？

精选留言 (9)

写留言

王天雨
2021-04-21

1、执行内存总量是动态变化的，最大是spark.executor.memory * spark.memory.fraction
本例中最大360M。
其次并发度N是固定不变，但是Executor中当前并行执行的任务数是小于等于N的，
上下限公式的计算是根据Executor中当前并行执行的并发度来计算的。...
展开

作者回复: 正解~ 赞~ 老弟对于执行内存的（1/N/2，1/N）理解得相当到位~

1 3

巧克力黑
2021-04-21

老师好!
实例1中
1、为什么提到“线程池大小设置为 1 是不可取的”？

2、假如spark.executor.cores设置成 1 ，有3个Task，串行。 ...
展开

作者回复: 1. 如果并发度是1，单个Executor就是纯粹的串行计算了，Spark“分布式”计算引擎的并行计算就失去了意义。即便还是可以有多个Executors，但是如果每个Executor在同一时间只有一个线程在工作，这种“分布式”计算的效率也是极其低下的。

2. 好问题，确实有这个隐患。不过呢，因为任务执行涉及的对象都是short-lived，也就是生命周期都很短，不像Storage Memory存储的缓存对象、生命周期很长。因此，执行任务涉及到的对象，多是放在年轻代，Minor GC的触发会频繁一些，回收效率也更高。不像long-lived objects，往往需要触发Full GC才能回收。因此，即便有还未回收的内存，也不影响任务执行，因为Minor GC频繁、且轻量，能够相对及时地把内存回收。

3

sky_sql
2021-04-21

老师好！麻烦问下 Shuffle 文件寻址有个参数spark.reducer.maxSizeInFlight 默认48m，这个buffer缓冲每次拉取48m数据。是Execution Memory剩余部分不够48m就会oom吗？这个和1/N的有啥关联？
展开

作者回复: 好问题，你说的是对的，这部分buffer也算作Execution Memory的一部分，也会记到Execution Memory的“账上”。因此，如果像你说的这种edge case，连48M都没有了，那拉数据的时候确实会OOM。这部分就没有1/N的关系了哈，每个task都是这么大的buffer，一旦不够，也就OOM了。

2

苏子浩
2021-04-27

老师，您好。关于文中在“case1: 数据倾斜”部分提到的“针对以这个案例为代表的数据倾斜问题，我们至少有 2 种调优思路：1. 消除数据倾斜，让所有的数据分片尺寸都不大于100MB；2. 调整 Executor 线程池、内存、并行度等相关配置，提高 1/N 上限到 300MB”。我有两点疑问：
（1）这里提到的数据分片尺寸100MB是怎么定的呢？为什么不是120MB呢？本例子中...
展开

作者回复: 好问题，我们一个一个来说。

1） 你说的没错，线程池大小为3的话，确实每个Task最多可以拿到120MB的内存。咱们原文里面建议分片大小100MB，目的其实就是留点富余，不要满打满地真的把分片大小弄到和上限一样大。

留富余的目的，其实就是为了任务执行的稳定性和可靠性。首先，这里面有你说的因素，也就是其他环节也都需要内存；再者，即便其他环节不消耗内存，我们也得留下富余。原因是，对于任务调度来说，作为开发者，我们不能直接掌控，任务总有先来后到，在N~ < N的时候，我们用N=3去计算每个Task分得的内存大小，就会有偏差，这个咱们原文也有讲。所以，我们在考虑每个分片数据量大小的时候，要尽量留些富余出来。

2）我们之所以用N去计算、去预估，有两个原因。第一个很简单，N~是不确定的，我们只知道N~ < N，但不知道N~具体是多少。再者，虽然由于调度系统的缘故，Task有先来后到，在任务调度初期N~确实小于N，但是随着执行的进行，N~总会追上来、等于N，而且这个过程其实很快。也就是在大部分时间，N~都是等于N的。另外，在你说的case里，只有增大Execution Memory，相应的1/N也才能跟着变大呀 😃

1

苏子浩
2021-04-27

老师，我想问一下，数据膨胀导致 OOM 的例子中，一定会出现OOM吗？既然Task1 能

获取到 300MB 的内存空间，那么挂起Task2线程和Task3线程，等待Task1内存释放，然后依次完成3个Task呢?

展开

作者回复: 你说的对，如果Spark可以保证依次运行这3个Tasks，确实不会OOM。不过，这里的关键是“依次”，当Executors线程池大于1的时候，Spark不能保证“依次”处理task。Task确实有先来后到，但是要是严格保证“依次”可就难了。

把spark.executor.cores置为1，就是咱们这讲举的极端的例子，Spark可以严格保证“依次”，不过就像咱们说的，它其实已经失去并行计算的意义了。

1

Fendora范东_

2021-04-21

1.task1首先运行的，拿到自己1/N发现还不够，就继续申请内存。task2/task3后面运行，发现可用内存满足下限，就跑去运行了，结果task1抢先申请到300M，task2,task3在运行时需要更多内存，不能得到满足，导致OOM。
2.driver端oom:
遇到过执行查询SQL，结果集太大，oom。通过调maxResultSize大小来解决。 ...

展开

作者回复: 1. 不完全对，task1之所以能拿到，是因为它“到的早”，它刚来的时候N=1，所以它最多可以拿到整个执行内存。task2、task3不行，是因为他们来得晚，他们到的时候，N至少等于2了，所以“每人”最多拿执行内存的一半，所以肯定会OOM。这里其实有点tricky，主要想考大家对于（1/N/2，1/N）的理解是不是足够透彻。

2. 对，driver OOM调整maxResultSize比较典型。学过这一讲，Executors内存调优可以做得更细致一些了，就不必每次都是单纯调大executor.memory~

1

Fendora范东_

2021-04-21

有个疑问
collect是把executor所有数据全部拉回来，但是他是一次性全放到driver端吗？如果是，那fetchsize的大小只是为了分批输出吗，是不是调的越大输出就越快了;如果不是，那就不应该出现driver端oom吧
按照文中分析来看，collect就是把所有数据一次性拉到driver端了吧?

展开

作者回复: collect是全量拉，你说的fetch size，我理解是单批次拉取的最大数据量，是用来限制网络吞吐的，因此这个参数和Driver是否OOM没有关系。Driver能承受的全量数据大小，由maxResultSize参数来决定，如果你拉的全量数据，大于maxResultSize，就会OOM。

1

aof
2021-05-05

老师，针对第一题，有一个疑问，一般来讲，如果数据分片/partition有3个，spark.executor.cores也大于等于3个的时候，很难会出现任务先来后到的情况吧？网络延迟会造成这种情况？我能想到的出现这种情况场景可能是：比如有100个分区待处理，spark.executor.cores为50，这样只能是先执行50个task，然后执行完task的core空闲下来之后，就可以执行pending的tasks了，那么这个时候就会出现task先来后到的情况了。

展开

苏子浩
2021-04-27

老师好，我有一下三个问题：
(i)关于Task获取内存方面中“每个线程分到的可用内存有一定的上下限，下限是 M/N/2，上限是 M/N，也就是属于[M/(2*N), M/N].”,同时注意到上述的M和N其实是随着Executor中task的状态动态变化的，根据前文提到的“执行内存总量M动态变化，由于Execution Memory可以占用Storage Memory以及抢占的优先级，所以ExecutionMemory的下限...

展开

作者回复: 都是很好的问题~ 没事，不用怕字数多，像这样的讨论多多益善~ 咱们一个一个来说。

i）先说执行内存的上限，你也说了，是Execution Memory、外加Storage Memory空闲的部分。再说内存抢占，这两部分内存的消耗，一定是优先消耗Execution Memory，然后再去抢占Storage Memory，而且，抢占的时候，还只能抢Storage Memory空闲的部分（Storage Memory的空间也是动态变化的，比如中间有缓存任务往Storage Memory里面灌数据）。这块其实可以想想黄小乙的占地协议，他必须先把自己的地种满，然后才能去抢人家的地，不能说，一上来，就把人家麻子的地给占了，那也不公平嘛！再者，关于任务的内存消耗，思考的很深入，挺好~ 我们要区分两个概念，一个，是Task可以获取到的内存上限、也就是1/N；一个，是Task计算对于内存的消耗需要。换句话说，一个是供给上限，一个是计算需求。内存的分配，是按照需求来分配的，也就是说，你需要多少，就逐步给你分配多少。并不是说，你的上限是120MB，Memory Manager一下子就把这120MB全部划给你、预留给你了。1/N只是说明，你最多可以拿这么多，但是你具体消耗多少，取决于你手里有多少农作物的种子。这就好比，你手里就1亩地的

种子，却想从黄小乙那里租赁10亩地，那黄小乙也不能同意呀！剩下的9亩地，不是白白浪费掉了么。

ii）好像已经把问题ii）回答了，哈哈，就是你说的那样，消耗300MB，还剩60MB。如果Task2到了、Task3还没到，就是你算的结果60MB/2 = 30MB。

iii）这个确实是个好问题，不过这个问题也暴露了你调度系统那一讲没吃透 😃，这部分其实要结合调度系统去理解。还记得DAGScheduler、TaskScheduler和SchedulerBackend吗？Executors的空闲状态，是由SchedulerBackend维护的，TaskScheduler拿到DAGScheduler拆解的Tasks之后，开始去SchedulerBackend那里尝试拿ResourceOffer，然后再从Tasks里面去挑，挑那些满足本地性级别要求的任务，然后再调度过去。你可以把DAGScheduler理解为任务执行的需求端，把SchedulerBackend理解成计算资源的供给端，而TaskScheduler就是中间的媒人、中介、撮合商。回答你的问题，首先，不是Driver一股脑把Tasks都分发过去，因为Tasks的执行需求，要先看SchedulerBackend有没有足够的资源供给。实际上，这个流程恰恰相反，是SchedulerBackend先提供ResourceOffer，然后TaskScheduler才能去挑Tasks，然后把任务调度过去。所以实际的调度情况，和你后者说的差不多，也就是你说的：“按照Executor的执行情况去发送Task”。任务调度确实比较难，除了前面的第5讲《调度系统》，还可以看看infoQ的这篇，这篇《权力的游戏》更详尽：https://www.infoq.cn/article/5aOHzQIaXX6NlHriLtSI